Introduction to Software Security

Karen Mercedes Goertzel [vita[1]]
Updated 2009-01-09

This article describes the need for and the challenges of building secure software, general principles of secure software development, and the key elements of a secure software life cycle process. The majority of this material is either adapted or excerpted from *Enhancing the Development Life Cycle to Produce Secure Software: A Reference Guidebook on Software Assurance* [DHS/DACS 08[2]].

Software's Vulnerability to Attack

What makes it so easy for attackers to target software is the virtually guaranteed presence of vulnerabilities, which can be exploited to violate one or more of the software's security properties. According to CERT, most successful attacks result from targeting and exploiting known, non-patched software vulnerabilities and insecure software configurations, many of which are introduced during design and code.

In their Report to the President titled *Cyber Security: A Crisis of Prioritization* [PITAC 05[9]], the President's Information Technology Advisory Committee summed up the problem of non-secure software as follows:

> Software development is not yet a science or a rigorous discipline, and the development process by and large is not controlled to minimize the vulnerabilities that attackers exploit. Today, as with cancer, vulnerable software can be invaded and modified to cause damage to previously healthy software, and infected software can replicate itself and be carried across networks to cause damage in other systems. Like cancer, these damaging processes may be invisible to the lay person even though experts recognize that their threat is growing. And as in cancer, both preventive actions and research are critical, the former to minimize damage today and the latter to establish a foundation of knowledge and capabilities that will assist the cyber security professionals of tomorrow reduce risk and minimize damage for the long term.

The security of software is threatened at various points throughout its life cycle, both by inadvertent and intentional choices and actions taken by "insiders"—individuals closely affiliated with the organization that is producing, deploying, operating, or maintaining the software, and thus trusted by that organization—and by "outsiders" who have no affiliation with the organization. The software's security can be threatened

- **during its development:** A developer may corrupt the software—intentionally or unintentionally—in ways that will compromise the software's dependability and trustworthiness when it is operational.
- **during its deployment (distribution and installation):** If those responsible for distributing the software fail to tamperproof the software before shipping or uploading, or transmit it over easily intercepted communications channels, they leave the software vulnerable to intentional or unintentional

1. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/1089-BSI.html (Goertzel, Karen Mercedes)
2. #dsy547-BSI_dacs
3. #dsy547-BSI_swvul
4. #dsy547-BSI_challenge
5. #dsy547-BSI_swa
6. #dsy547-BSI_princ
7. #dsy547-BSI_what
8. #dsy547-BSI_integrat
9. #dsy547-BSI_pitac

corruption. Similarly, if the software's installer fails to "lock down" the host platform, or configures the software insecurely, the software is left vulnerable to access by attackers.

- **during its operation:** Once COTS and open source software has gone operational, vulnerabilities may be discovered and publicized; unless security patches and updates are applied and newer supported versions (from which the root causes of vulnerabilities have been eliminated) are adopted, such software will become increasingly vulnerable. Non-commercial software and open source software (OSS) may also be vulnerable, especially as it may manifest untrustworthy behaviors over time due to changes in its environment that stress the software in ways that were not anticipated and simulated during its testing. Any software system that runs on a network-connected platform has its vulnerabilities exposed during its operation. The level of exposure will vary depending on whether the network is public or private, Internet-connected or not, and whether the software's environment has been configured to minimize its exposure. But even in highly controlled networks and "locked down" environments, the software may be threatened by malicious insiders (users, administrators, etc.).
- **during its sustainment:** If those responsible for addressing discovered vulnerabilities in released software fail to issue patches or updates in a timely manner, or fail to seek out and eliminate the root causes of the vulnerabilities to prevent their perpetuation in future releases of the software, the software will become increasingly vulnerable to threats over time. Also, the software's maintainer may prove to be a malicious insider, and may embed malicious code, exploitable flaws, etc., in updated versions of the code.

Both research and real-world experience indicate that correcting weaknesses and vulnerabilities as early as possible in the software's life cycle is far more cost-effective over the lifetime of the software than developing and releasing frequent security patches for deployed software.

The Challenge of Building Secure Software

External faults that threaten the software's dependable operation are seen as a security issue when (1) the faults result from malicious intent or (2) the faults, regardless of their cause, make the software vulnerable to threats to its security. According to Bruce Schneier in *Beyond Fear* [Schneier 06[10]], "Security is about preventing adverse consequences from the intentional and unwarranted actions of others."

Enhancing the Development Life Cycle to Produce Secure Software [DHS/DACS 08[11]] defines secure software as follows:

> To be considered secure, software must exhibit three properties:
>
> 1. **Dependability:** Dependable software executes predictably and operates correctly under all conditions, including hostile conditions, including when the software comes under attack or runs on a malicious host.
> 2. **Trustworthiness:** Trustworthy software contains few if any vulnerabilities or weaknesses that can be intentionally exploited to subvert or sabotage the software's dependability. In addition, to be considered trustworthy, the software must contain no malicious logic that causes it to behave in a malicious manner.
> 3. **Survivability (also referred to as "Resilience"):** Survivable—or resilient—software is software that is resilient enough to (1) either resist (i.e., protect itself against) or tolerate (i.e., continue operating dependably in spite of) most known attacks plus as many novel attacks as possible, and (2) recover as quickly as possible, and with as little damage as possible, from those attacks that it can neither resist nor tolerate.

The objective of secure software development is to design, implement, configure, and sustain software systems in which security is a necessary property from the beginning of the system's life cycle (i.e., needs and requirements definition) to its end (retirement). Experience has taught that the most effective way to achieve secure software is for its development life cycle processes to rigorously conform to secure

10. #dsy547-BSI_bruce
11. #dsy547-BSI_dacs

development, deployment, and sustainment principles and practices. Organizations that have adopted a secure software development life cycle (SDLC) process have found almost immediately upon doing so that they have begun finding many more vulnerabilities and weaknesses in their software early enough in the SDLC that they are able to eradicate those problems at an acceptable cost. Moreover, as such secure practices become second nature over time, these same developers start to notice that they seldom introduce such vulnerabilities and weaknesses into their software in the first place.

Software Assurance

The main objective of software assurance is to ensure that the processes, procedures, and products used to produce and sustain the software conform to all requirements and standards specified to govern those processes, procedures, and products. Software security and secure software are often discussed in the context of software assurance. Software assurance in its broader sense refers to the assurance of any required property of software. For software practitioners at the National Aeronautics and Space Administration (NASA), software assurance refers to the assurance of safety as a property of software. Similarly, in other communities, software assurance may refer to assurance of reliability or quality. In the context of this article, software assurance is concerned with assuring the *security* of software.

An increasingly agreed-upon approach for assuring the security of software is the software security assurance case, which is intended to provide justifiable confidence that the software under consideration (1) is free of vulnerabilities; (2) functions in the "intended manner," and this "intended manner" does not compromise the security or any other required properties of the software, its environment, or the information it handles; and (3) can be trusted to continue operating dependably under all anticipated circumstances, including anomalous and hostile environmental and utilization circumstances—which means that those who build the software need to anticipate such circumstances and design and implement the software to be able to handle them gracefully. Such circumstances include

- the presence of unintentional faults in the software and its environment
- the exposure of the operational software to accidental events that threaten its security
- the exposure of the software to intentional choices or actions that threaten its security during its development, deployment, operation, or sustainment

Software is more likely to be assurably secure when security is a key factor in the following aspects of its development and deployment:

- **development principles and practices:** The practices used to develop the software and the principles that governed its development are expressly intended to encourage and support the consideration and evaluation of security in every phase of the software's development life cycle. Some secure development principles and practices for software are suggested later in this article.
- **development tools:** The programming language(s), libraries, and development tools used to design and implement the software are evaluated and selected for their ability to avoid security vulnerabilities and to support secure development practices and principles.
- **testing practices and tools:** The software is expressly tested to verify its security, using tools that assist in such testing.
- **acquired components:** Commercial off-the-shelf (COTS) and OSS components are evaluated to determine whether they contain vulnerabilities, and if so whether the vulnerabilities can be remediated through integration to minimize the risk they pose to the software system.
- **deployment configuration:** The installation configuration of the software minimizes the exposure of any residual vulnerabilities it contains.
- **execution environment:** Protections are provided by the execution environment that can be leveraged to protect the higher level software that operates in that environment.
- **practitioner knowledge:** The software's analysts, designers, developers, testers, and maintainers are provided with the necessary information (e.g., through training and education) to give them sufficient

security awareness and knowledge to understand, appreciate, and effectively adopt the principles and practices that will enable them to produce secure software.

General Principles of Secure Software Development

The following principles should guide the development of secure software, including all decisions made in producing the artifacts at every phase of the software life cycle.

Minimize the number of high-consequence targets. The software should contain as few high-consequence targets (critical and trusted components) as possible. High-consequence targets are those that represent the greatest potential loss if the software is compromised and therefore require the most protection from attack. Critical and trusted components are high-consequence because of the magnitude of impact if they are compromised. *(This principle contributes to trustworthiness and, by its implied contribution to smallness and simplicity, also to dependability.)*

Don't expose vulnerable and high-consequence components. The critical and trusted components the software contains should not be exposed to attack. In addition, known vulnerable components should also be protected from exposure because they can be compromised with little attacker expertise or expenditure of effort and resources. *(This principle contributes to trustworthiness.)*

Deny attackers the means to compromise. The software should not provide the attacker with the means by which to compromise it. Such "means" include exploitable weaknesses and vulnerabilities, dormant code, backdoors, etc. Also, provide the ability to minimize damage, recover, and reconstitute the software as quickly as possible following a compromising (or potentially compromising) event to prevent greater compromise. In practical terms, this will require building in the means to monitor, record, and react to how the software behaves and what inputs it receives. *(This principle contributes to dependability, trustworthiness, and resilience.)*

Always assume "the impossible" will happen. Events that seem to be impossible rarely are. They are often based on an expectation that something in a particular environment is highly unlikely to exist or to happen. If the environment changes or the software is installed in a new environment, those events may become quite likely. The use cases and scenarios defined for the software should take the broadest possible view of what is possible. The software should be designed to guard against both likely and *un*likely events.

Developers should make an effort to recognize assumptions they are not initially conscious of having made and should determine the extent to which the "impossibilities" associated with those assumptions can be handled by the software. Specifically, developers should always assume that their software will be attacked, regardless of what environment it may operate in. This includes acknowledgement that environment-level security measures such as access controls and firewalls, being composed mainly of software themselves (and thus equally likely to harbor vulnerabilities and weaknesses), can and will be breached at some point, and so cannot be relied on as the sole means of protecting software from attack.

Developers who recognize the constant potential for their software to be attacked will be motivated to program defensively, so that software will operate dependably not only under "normal" conditions but under anomalous and hostile conditions as well. Related to this principle are two additional principles about developer assumptions.

1. **Never make blind assumptions.** Validate every assumption made by the software or about the software *before* acting on that assumption.
2. **Security software is not the same as secure software.** Just because software performs information security-related functions does not mean the software itself is secure. Software that performs security functions is just as likely to contain flaws and bugs as other software. However, because security functions are high-consequence, the compromise or intentional failure of such software has a significantly higher potential impact than the compromise or failure of other software.

What the Software Practitioner Needs to Know

The main characteristics that discriminate the developer, tester, integrator, and sustainer of secure software from those of non-secure software are awareness, intention, and caution. A software professional who cares about security and acts on that awareness will recognize that software vulnerabilities and weaknesses can originate at any point in the software's conception or implementation, from inadequate requirements, to poor design and implementation choices, to inadvertent coding errors or configuration mistakes.

The security-aware software professional knows that the only way these problems can be avoided is through well-informed and intentional effort: requirements analysts must understand how to translate the need for software to be secure into actionable requirements, designers must recognize choices that conflict with secure design principles, and programmers must follow secure coding practices and be cautious about avoiding coding errors and finding and removing the bugs they were unable to avoid. Software integrators must recognize and strive to reduce the security risk associated with vulnerable components (whether custom-built, COTS, or open source), and must understand the ways in which those modules and components can be integrated to minimize the exposure of any vulnerabilities that cannot be eliminated.

The main reason for adding security practices throughout the SDLC is to establish a software life cycle process that codifies both caution and intention.

Creating a secure development community using collaboration technologies and a well-integrated development environment promotes a continuous process of improvement and a focus on secure development life cycle principles and practices that will result in the ongoing production of more dependable, trustworthy, survivable software systems.

Integrating Security into the Software Life Cycle

"Security enhancement" of the SDLC process mainly involves the adaptation or augmentation of existing SDLC activities, practices, and checkpoints, and in a few instances, it may also entail the addition of new activities, practices, or checkpoints. In a very few instances, it may also require the elimination or wholesale replacement of certain activities or practices that are known to obstruct the ability to produce secure software.

The key elements of a secure software life cycle process are

- security criteria in all software life cycle checkpoints (both at the entry of a life cycle phase and at its exit)
- adherence to secure software principles and practices
- adequate requirements, architecture, and design
- secure coding practices
- secure software integration/assembly practices
- security testing practices that focus on verifying the dependability, trustworthiness, and sustainability of the software being tested
- secure distribution and deployment practices and mechanisms
- secure sustainment practices
- supportive tools
- secure software configuration management systems and processes
- security-knowledgeable software professionals
- security-aware project management
- upper management commitment to production of secure software

Organizations can insert secure development practices into their software life cycle process either by adopting a codified secure software development methodology, such as those discussed in Section 3.6 of

Enhancing the Development Life Cycle to Produce Secure Software [DHS/DACS 08[12]], and the SDLC Process[13] content area of Build Security In, or through the evolutionary security enhancement of their current practices, as described in Sections 4-10 of *Enhancing the Development Life Cycle to Produce Secure Software* and in the Best Practices[14] and Knowledge[15] sections of Build Security In.

These, as well as the other Best Practices, Knowledge, and Tools[16] articles on Build Security In support organizations in making progress toward achieving these goals. Those responsible for ensuring that software and systems meet their security requirements throughout the development life cycle should review, select, and tailor BSI guidance as part of normal project management activities. Additional Resources[17] on BSI and the references below provide additional, experience-based practices and lessons learned that development organizations need to consider.

References

[DHS/DACS 08]	Goertzel, Karen, Theodore Winograd, et al. for Department of Homeland Security and Department of Defense Data and Analysis Center for Software. *Enhancing the Development Life Cycle to Produce Secure Software*[18]*: A Reference Guidebook on Software Assurance*, October 2008.
[PITAC 05]	President's Information Technology Advisory Committee. *Cyber Security: A Crisis of Prioritization*[19]*: Report to the President*. National Coordination Office for Information Technology Research and Development, February 2005.
[Schneier 06]	Schneier, Bruce. *Beyond Fear*. Heidelberg, Germany: Springer-Verlag, 2006.

12. #dsy547-BSI_dacs
13. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/sdlc.html (SDLC Process)
14. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices.html (Best Practices)
15. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge.html (Knowledge)
16. http://buildsecurityin.us-cert.gov/bsi/articles/tools.html (Tools)
17. http://buildsecurityin.us-cert.gov/bsi/resources.html (Additional Resources)